C＋＋によるオブジェクト指向プログラミングと手続き型プログラミングの比較

Comparison between Object-Oriented Programming and Procedural Programming in C++

川　口　順　功

Junkou KAWAGUCHI

（平成8年4月9日受理）

C＋＋は，手続き型プログラミング言語であるCを部分集合として持つ，オブジェクト指向プログラミングをサポートするプログラミング言語である[1]．関数呼び出しを中心とする手続き型プログラミング（Procedural Programming）と，クラスとそれから生成されるオブジェクトを中心とするオブジェクト指向プログラミング（Object-Oriented Programming）には，プログラムの表現方法やプログラミングの考え方に大きな違いが見られるが，C＋＋は両方に対応できる言語である．したがって，それらの違いを調べるには最適の言語である．

本稿では，C＋＋[2]により手続き型プログラミングとオブジェクト指向プログラミングで具体的なプログラムを作成し，両者をプログラムの表現性（わかりやすさ）と拡張性の点で比較し，その違いについて考察した．簡単な処理のプログラムであるにも拘わらず，両者の間には予想した以上の違いが見られた．手続き型プログラミングでは，簡単な機能の追加に伴うプログラムの変更が全体におよび，プログラムが複雑になってしまった．一方，オブジェクト指向プログラミングでは，同じ条件でのプログラムの変更がクラスの変更だけで対応でき，プログラムの表現性においては変更前とほぼ同じであった．そこで，このようなことの背景となったオブジェクト指向プログラミングのためのいくつかの機能に注目し，プログラムの表現性と拡張性におけるオブジェクト指向プログラミングの優位性について考察した．

1. はじめに

オブジェクト指向プログラミングは，SmalltalkやC＋＋などのオブジェクト指向言語の実用化に伴い，手続き型プログラミングなどと比較されるようになってきたプログラミング技法の一つである．これについて，C＋＋の設計者であるB.Stroustrupも，その著書の中で，「オブジェクト指向プログラミングは，プログラミング技法―ある問題の集合に対して“良い”プログラムを書くためのパラダイム―である．」[3]と言っている．処理の流れを中心に考える手続き型プログラミングでは，関数に引数を渡し，その関数からの戻り値や関数内での副作用(side effect)を使うことによってプログラムを作成する．一方，オブジェクト指向プログラミングでは，データ抽象型としてのクラスを作成し，それから生成されるオブジェクトを操作することによってプログラムを作成する．この相違により，プログラムの表現方法やプログラミングの考え方が大きく違ってくる．これらが両者の表現性や拡張性とどのように関係しているかを調べるのは，意義のあることに思われる．なぜなら，プログラムの表現性と拡張性は，“良い”プログラムの大きな条件と考えられるからである．

オブジェクト指向プログラミングでは，その名のとおりオブジェクト(object)が中心となる．オブジェクトは，ある状態を共有するメソッド（method）と呼ばれる操作の集まりである．[4]そのオブジェクト内の状態は，外部からは隠蔽され，オブジェクトの操作によってのみ変わりうる．オブジェクトの原型になるのがクラスである．オブジェクトは，クラスより一つのインスタンス(instanse)として生成される．したがって，オブジェクトへのインターフェイスやその振る舞いは，クラスの定義によって決定される．オブジェクトの内部状態は，一般的には外部からはアクセスできないプライベート（private）なデータメンバで表す．このデータメンバにアクセスして状態を変えるのがメンバ関数（メソッド）である．外部からはオブジェクトに対するメッセージ(message)として，これらのメンバ関数を呼び出すことによってのみオブジェクトを操作できる．ただし，外部から呼び出せるメンバ関数はパブリック(public)なものだけである．このように，データとそれを操作する関数を一つの集合としてまとめることをカプセル化（encapsulation）といい，外部からプライベートなデータメンバをアクセスできないようにすることを情報遮蔽（data hiding）という．これらは，データ抽象の概念からくるものであり，オブジェクト指向プログラミングを特徴づけるものの一つである．

C＋＋では，オブジェクト指向プログラミングの基本となるのがクラスの概念である．一つのアプリケーションプログラムを作成するとき，クラスはそのアプリケーションの概念の一つを表すものとなる．したがって，オブジェクト指向プログラミングでは，クラスによってアプリケーションの概念をうまく表現できるかどうかが重要となる．クラスの基本機能（クラスの継承など）によりクラス間の相互作用をうまく表現できたオブジェクト指向プログラミングによるプログラムは，目的のアプリケーションを容易に実現できるだけでなく，表現性と拡張性の点において手続き型プログラミングのものよりもはるかに優位になることが期待できる．

オブジェクト指向プログラミングでは，データのカプセル化，情報隠蔽，多重定義など手続き型プログラミングには見られない，プログラムの表現性や拡張性にとって有効な機能が利用できる．本稿では，これらの有効な機能に注目し，C＋＋による手続き型プログラミングとオブジェクト指向プログラミングとを具体的なプログラムによって比較し，両者のプログラムの表現性と拡張性についての違いを調べ，これらに対する後者の優位性について考察した．

2．オブジェクト指向プログラミングとクラス

オブジェクト指向プログラミングではクラスの定義が基本となる．クラスは，既存の型と同じ完全な仕様を持つように定義される利用者定義の型となる．クラスの要素はメンバと呼ばれ，メンバにはデータメンバとメンバ関数がある．C＋＋では，クラスは次のように定義される．

```
class クラス名 {
  private :
        メンバ
  protected :
        メンバ
  public :
        メンバ
} ;
```

private 部で指定されるメンバは，クラスのメンバ関数とそのクラスでフレンド指定された関数からのみ参照できる．protected 部で指定されるメンバは，このクラスの派生クラスから参照できること以外は private 部のものと同じである．public 部のメンバは，メンバ関数を含めどの関数からも参照できる．したがって，ここで定義されるメンバによってクラスのユーザインターフェイスが決まる．メンバ関数の本体は，クラスの内部でも外部でも定義できる．内部で定義したものはインライン（inline）関数となる[5] ここでは，具体的なプログラムを例にとり，手続き型プログラミングとオブジェクト指向プログラミングの違いをプログラムの表現性と拡張性の観点から検証し，クラスの基本機能について考察する．

タートル・グラフィックスのプログラムの例で考察する．タートル・グラフィックスでは，タートル（turtle：亀）を指定された方向に指定された長さだけ動かし，その軌跡によっていろいろな図形を描く．このタートル・グラフィックスによって図２－１の正三角形を描くプログラムを例にとる．手続き型プログラミングで表現したものがプログラム２－１[6] オブジェクト指向プログラミングで表現したものがプログラム２－２[7]である．まず，この両者を比較し，その違いについて考察する．

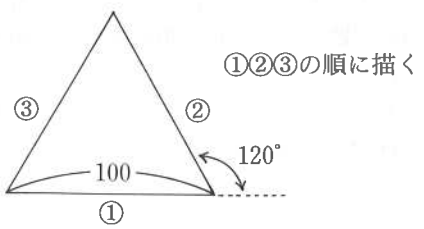

図２－１　１つの正三角形

タートル・グラフィックスのために，基本的なデータ構造（以下，基本データと呼ぶ）となる変数としてタートルの位置（X座標とY座標）と向き（角度：X座標の正の向きを０°，反

時計回りの向きを正とする）を表すもの，基本的な関数（以下，基本関数と呼ぶ）としてタートルの軌跡を描くものとタートルの方向変換のためのものを用意することにする．また，タートルを位置づける関数は任意とする．

手続き型プログラミング（プログラム２－１）においては，タートルの位置（X，Y）と向き（Angle）をグローバル（global：大域）構造体 Tdata のデータメンバとし，タートルの位置づけ，タートルの軌跡，タートルの方向変換をそれぞれ関数 initurt，goahead，turn で表現している．main 関数では，これらの関数を使って，正三角形を次のような６つの命令によって描いている．

① initurt(100，200，0)　・・・・　タートルを(100，200)に位置づけて，向きを 0°にする
② goahead(100)　・・・・　タートルを 100 だけ進め，線を描く
③ turn(120)　・・・・　タートルの向きを 120°変える
④ goahead(100)　・・・・　タートルを 100 だけ進め，線を描く
⑤ turn(120)　・・・・　タートルの向きを 120°変える
⑥ goahead(100)　・・・・　タートルを 100 だけ進め，線を描く

これらの命令は正三角形を描くための処理手順そのものであり，手続き型プログラミングの典型的なプログラムスタイルである．基本データをグローバル構造体として定義しているのは，どの関数からも引数としないで参照できるようにするためである．この構造体をローカル（local：局所）にして，各関数に引数として渡す方法も考えられる．（この方法によるプログラムは後で示す．）ここでは，基本データを引数としないオブジェクト指向プログラミングとの比較のためにグローバルとしてある．手続き型プログラミングのプログラミングパラダイムは，「必要な手続きを決定し，見つけられる最良のアルゴリズムを使う」[8]ことである．一般的には，データを関数（モジュールという表現が適切かもしれないが，Ｃ＋＋ではモジュール＝関数であるので，以下も関数と呼ぶ）に引数として渡し，その結果を利用してプログラムを作成する．結果とは，関数の返す値や関数内での副作用のことである．このプログラムでは，基本データをグローバル変数で定義してあるので，基本関数への引数とはなっていないが，実質的には引数である．

では，このプログラムの表現性と拡張性について見てみる．プログラムの表現性という点では，プログラムの処理内容が簡単であることと，プログラムの各命令が正三角形を描く直接の命令と対応しているので，特に問題はない．正三角形を描く手順がそのままプログラムとなっているので，わかりやすいプログラムといえる．

次に，拡張性の点から見てみる．タートル・グラフィックスの基本データがグローバルであることにより，上記①の initurt は特に呼び出す必要はなく，次のように各変数に直接アクセスして値を設定できる．

```
turt.X＝100.0；turt.Y＝200.0；turt.Angle＝0.0；
```

このように基本データの変数に直接アクセスできることは，プログラミングが柔軟にできる反面，プログラムのバグの原因になったり，機能変更に伴うプログラムの変更という面から問題

になる場合がある．例えば，プログラムにおける基本データの変更が生じた場合に，それらが使われている箇所が特定できないために，プログラム全体を見直さなければならない．この問題は，このような基本データの使い方をしないという約束によって避けられるように思われるが，すべてのプログラマがこの約束を守るという保証がない以上解決されるとはいえない．また，このように基本データの変数に直接アクセスできるということは，上記の②～⑥の命令の途中（例えば③と④の間）である関数を呼び出した場合，これらの変数が変更されている可能性があることを意味する．実際に，プログラムが大規模になり複数のプログラマによってプログラムを作成する時に，このようなことが発生することは十分予測できることである．このプログラムのように，基本データをグローバル変数として扱う場合のイメージは，図２－２のようになる．このようにデータを共有する場合，ある関数で誤ったデータの使い方をしてしまうと，すべての関数にとってのデータの完全性が失われてしまうことになる．このようなことは，特にプログラムを変更するときに発生しやすく，基本データと基本関数が実質的な引数として結びついているにも拘わらず，それらが一つのクローズされた集合体としてまとまっていないことに原因があると考えられる．以上の点から拡張性の面ではかなり問題があるといえる．

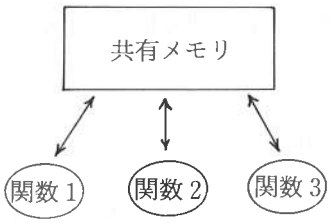

図２－２　共有メモリのイメージ

では，基本データを局所変数として定義し，それらを引数として渡す方法でプログラムを作成すれば，上述した問題は解決できるであろうか．このプログラムをプログラム２－１－２に示す．基本データの値を連続して使うため，引数は参照（アドレス）渡しとなる．タートルの位置づけの関数 initurt は，必要がないため削除してある．位置づけは，局所変数の基本データに値を直接設定して行っている．このプログラムによって，他の関数で基本データを変更されるという問題は解決できる．しかし，基本データを複数の関数の中で定義し，その有効範囲内で自由に使えるため，拡張性からの問題の根本的な解決にはならない．また，基本データを引数とすること自体にも問題がある．基本データの変更が生じた場合，プログラムの見直しが全体に及ぶからである．

オブジェクト指向プログラミング（プログラム２－２）においては，まず基本データと基本関数を一つのクラスとしてまとめ，次のように定義している．（個々の変数名や関数名の対応はプログラム２－１と同じである．）

```
Class Turt {
  private:
    X, Y, Angle;
```

```
    public :
      Turt() ;
      Turt(float, float, float) ;
      goahead(float) ;
      turn(float) ;
} ;
```

タートルの位置（X，Y）とその向き（Angle）をprivate部のデータメンバとし，タートルの軌跡（goahead）と方向変換（turn）をpublic部のメンバ関数としている．この定義によって基本データとなる変数には，クラスの外部からはアクセスできなくなる．このように，クラスによって基本データとそのデータを操作する基本関数を1つの集合にまとめることをカプセル化といい，基本データに外部からアクセスできないことを情報隠蔽という．オブジェクト指向プログラミングのプログラミングパラダイムは，「どのクラスを必要とするか決定せよ．各クラスについてその演算の全集合を与えよ．継承を使って共通性を明示的にせよ．」[9]である．したがって，オブジェクト指向プログラミングでは，このようなクラスの定義が基本となり，定義されたクラスは一つの利用者定義の新しい型として使える．ここでは，クラスTurt型としてkameというオブジェクトを次のように定義している．

```
Turt kame(100, 200, 0) ;                    (1)
```

この定義によって，kameというオブジェクトが実体として存在するようになり，このオブジェクトの状態を表すデータメンバのコピーが記憶領域に確保され，引数の値によって初期化される．以下，main関数において，このオブジェクトに対する次のような命令（メッセージ）によって正三角形を描いている．

```
① kame.goahead(100)     ····  kame(タートル)を100だけ進め，線を描く
② kame.turn(120)        ····  kame(タートル)の向きを120°変える
③ kame.goahead(100)     ····  kame(タートル)を100だけ進め，線を描く
④ kame.turn(120)        ····  kame(タートル)の向きを120°変える
⑤ kame.goahead(100)     ····  kame(タートル)を100だけ進め，線を描く
```

プログラム2－1のinitturに対応する命令は，(1)でkameを定義するときのコンストラク（constructor）で実行される．コンストラクタはクラス名と同じメンバ関数で定義される．このプログラムでは2つのコンストラクタが多重定義されている．これ以外のmain関数における各命令は，形式的にはプログラム2－1と同じである．したがって，プログラム2－1と同程度にわかりやすいプログラムといえる．しかし，内容的に見ると両者には大きな違いがある．手続き型プログラミングの各命令は，正三角形を描く手順としてのコンピュータに対するものと見なせる．一方，オブジェクト指向プログラミングでは，kameというオブジェクトを1つの図形を描くタートル（ロボット）と考えると，各命令はこのタートルに対する命令と見なせる．つまり，指示通りに動くタートルに口頭で命令して，正三角形を描かしているのと同じことに

なる．すると，これらの命令は現実におけるごく自然な命令であり，このオブジェクトを使ったプログラムは，まさに現実を写像したものといえる．

次に，拡張性の点から見てみる．プログラム２－１と形式的には同じプログラムに見えるが，内容的には大きな違いがある．これらの違いについて考察してみる．第一に，基本データは，基本関数（コンストラクタも含む）からしかアクセスできないことである．これによって，基本データの変更があっても，基本関数へのインターフェイスが変わらない限り，このクラス定義を変更するだけでよいことになる．このことは，プログラムの基本データの変更に伴うプログラムの変更が，このクラスのみに限られることを意味する．第二に，基本データの初期化が，コンストラクタによってオブジェクト定義の時点で自動的に行われることである．手続き型プログラミングでは，プログラマが意図的にしない限り基本データは初期化されない．初期化されないデータは，プログラムのバグの原因になる．オブジェクト指向プログラミングでは，クラスのメンバ関数としてコンストラクタを定義することによって，基本データの必要な初期化は自動的に行われる．したがって，適切なコンストラクタの定義によって，初期化のミスによるバグは完全に回避できる．また，コンストラクタに対応して，デストラクタ(destructor)も定義できる．これは，オブジェクトが消滅する直前で実行される．通常はヒープ領域から確保した領域を解放するときなどに定義する．これら一連の操作により，基本データの一貫性が保たれることになる．以上の点から，プログラム２－１と比較して拡張性がかなり高いと判断できる．これは，クラスのカプセル化や情報隠蔽の有効性に基づくものといえる．

```
// １つの正三角形 (turt21.c)
#include <stdio.h>
#include <graphics.h>
#include <math.h>
double PI=3.14159265;
struct Tdata {
        float X,Y,Angle;
} turt;
void initurt(float x, float y, float a)
{
        turt.X=x; turt.Y=y; turt.Angle=a;
}
void goahead(float d)
{
        moveto(turt.X,turt.Y);
        turt.X+=d*cos(PI*turt.Angle/180);
        turt.Y-=d*sin(PI*turt.Angle/180);
        lineto(turt.X,turt.Y);
}
void turn(float a)
{
        turt.Angle+=a;
}
void main(void)
{
        int gdrive=DETECT,gmode;
        initgraph(&gdrive,&gmode,"a:¥¥TC¥¥BGI");
        cleardevice();
        setcolor(BLUE);
        initurt(100,200,0);
        goahead(100);
        turn(120);
```

```
        goahead(100);
        turn(120);
        goahead(100);
        getch();
        closegraph();
}
```

プログラム２－１　１つの正三角形（手続き型１）

```
// １つの正三角形　(turt212.c)
#include <stdio.h>
#include <graphics.h>
#include <math.h>
double PI=3.14159265;
void goahead(float* X, float* Y, float Angle, float d)
{
        moveto(*X,*Y);
        *X+=d*cos(PI*Angle/180);
        *Y-=d*sin(PI*Angle/180);
        lineto(*X,*Y);
}
void turn(float* Angle, float a)
{
        *Angle+=a;
}
void main(void)
{
        int gdrive=DETECT,gmode;
        struct Tdata {
                float x,y,a;
        } turt;
        initgraph(&gdrive,&gmode,"a:¥¥TC¥¥BGI");
        cleardevice();
        setcolor(BLUE);
        turt.x=100; turt.y=200; turt.a=0;
        goahead(&turt.x,&turt.y,turt.a,100);
        turn(&turt.a,120);
        goahead(&turt.x,&turt.y,turt.a,100);
        turn(&turt.a,120);
        goahead(&turt.x,&turt.y,turt.a,100);
        getch();
        closegraph();
}
```

プログラム２－１－２　１つの正三角形（手続き型２）

```
// １つの正三角形 (turt22.cpp)
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>
const double PI=3.14159265;
class Turt {
  private:
        float X,Y,Angle;
  public:
```

```
        Turt();
        Turt(float,float,float);
        void goahead(float);
        void turn(float);
};
Turt::Turt()
{
        X=320; Y=200; Angle=0;
}
Turt::Turt(float x, float y, float a)
{
        X=x; Y=y; Angle=a;
}
void Turt::goahead(float d)
{
        moveto(X,Y);
        X+=d*cos(PI*Angle/180);
        Y-=d*sin(PI*Angle/180);
        lineto(X,Y);
}
void Turt::turn(float a)
{
        Angle+=a;
}
void main(void)
{
        int gdrive=DETECT,gmode;
        initgraph(&gdrive,&gmode,"a:¥¥TC¥¥BGI");
        Turt kame(100,200,0);
        cleardevice();
        setcolor(BLUE);
        kame.goahead(100);
        kame.turn(120);
        kame.goahead(100);
        kame.turn(120);
        kame.goahead(100);
        getch();
        closegraph();
}
```

プログラム２－２　１つの正三角形（オブジェクト指向）

次に，図２－３の２つの正三角形を図の番号順に描くプログラムについて考える．単純に２つの正三角形を交互に描いていくだけであるが，手続き型プログラミングとオブジェクト指向

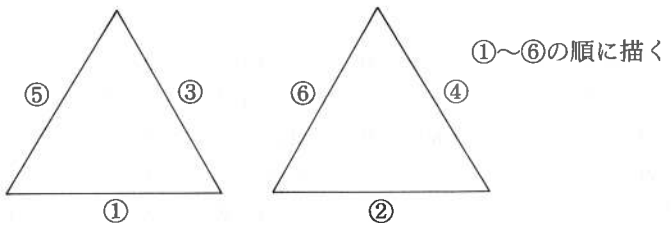

図２－３　２つの正三角形

プログラミングでは内容的に大きな違いがでてくる．手続き型プログラミングで表現したものをプログラム２－３に，オブジェクト指向プログラミングで表現したものをプログラム２－４に示す．基本データや基本関数は前と同じである．

手続き型プログラミング（プログラム２－３）では，２つの正三角形を描くために，それぞれの基本データのコピーを局所変数として，次のように用意しなければならない．

```
struct Tdata T1, T2;
```

変数名Ｔ１が左の正三角形を，変数名Ｔ２が右の正三角形を描くためのものである．正三角形の１辺を描くごとにグローバル変数の基本データturt（X，Y，Angle）をこれらの局所変数に保存し，同じ三角形の次の辺を描くとき，この保存した値でタートルを位置づけしなければならない．

プログラムの表現性の点から見てみる．プログラム２－１と比較してかなりわかりにくくなっている．辺を描くという直接の命令以外の命令が入り込み，プログラム全体の処理が説明なしにはわからなくなってしまう可能性がある．このようなケースは，処理の追加に伴うプログラムの変更において，プログラムが少しずつわかりにくくなる手続き型プログラミングの典型的な例といえる．したがって，プログラム２－３は，処理の単純さに比べわかりやすいプログラムとはいえない．

次に，拡張性の点で見てみる．基本データのコピーが利用者のプログラムの中で定義されることは，プログラム２－１－２と同じ理由で問題がある．しかも，このプログラムでは基本データの変数自体を利用者のプログラムの中で使わざる得ない状況であることより，拡張性はさらに悪くなっているといえる．また，正三角形の１辺を描くたびに変数を保存し，次の辺を描くときタートルを位置づけることは，処理効率の点からも明らかによくない．これらの問題点は，このプログラムの問題が２つの正三角形に対応するそれぞれの基本データを必要とすることに原因があるといえる．しかし，これらの問題点を解決するには，基本データとなるグローバル変数を２組用意するだけではなく，さらにそれぞれに対応する基本関数を別々に用意する必要がある．これは，正三角形が３つ，４つと増える可能性を考えれば根本的な解決法にならないことは明らかである．

基本データを局所変数としたプログラムは，プログラム２－３－２のようになる．プログラム２－３との大きな違いは，基本データの保存やタートルの位置づけなどの命令が必要ないことである．したがって，プログラムの表現性の点からは，プログラム２－１－２とそんなにかわらない．プログラムを次のように書けば，このことはより明確にわかる．

```
goahead(&T1.X,&T1.Y,T1.Angle,100); goahead(&T2.X,&T2.Y,T2.Angle,100);
turn(&T1.Angle,120);                turn(&T2.Angle,120);
goahead(&T1.X,&T1.Y,T1.Angle,100); goahead(&T2.X,&T2.Y,T2.Angle,100);
turn(&T1.Angle,120);                turn(&T2.Angle,120);
goahead(&T1.X,&T1.Y,T1.Angle,100); goahead(&T2.X,&T2.Y,T2.Angle,100);
```

問題点は，基本関数の呼び出しがどちらの三角形のものかが，引数の変数名からしか判断で

きないことである．以上のことから，プログラムの表現性の点からは，基本データをグローバル変数とするより局所変数としてプログラムを表現した方がよいことがわかる．拡張性の点からは，プログラム２－１－２と同じように基本データが利用者のプログラムで直接定義され，引数として使われることから問題がある．

オブジェクト指向プログラミング（プログラム２－４）では，プログラム２－２と同じようにクラス Turt を定義し，次のように２つのオブジェクトを定義している．

```
Turt kame(100, 200, 0), kame2(320, 200, 0);
```

kame が左の正三角形，kame２が右の正三角形を描くためのオブジェクトである．この定義によって基本データとなる変数が，それぞれのオブジェクトに対応して確保され，それぞれの引数の値により初期化される．後はそれぞれのオブジェクトにメッセージを送り，目的の正三角形を描くだけである．kame に対するメッセージが左の正三角形，kame２に対するメッセージが右の正三角形を描くためのものであるということさえ念頭に入れれば，プログラムの表現性という点ではプログラム２－２とそれほど変わらない．手続き型プログラミングのプログラム２－１とプログラム２－３の違いを考えると，この点は大いに注目する必要がある．プログラムを次のように書けばさらにわかりやすいものになる．

```
kame.goahead(100);  kame2.goahead(100);
kame.turn(120);     kame2.turn(120);
kame.goahead(100);  kame2.goahead(100);
kame.turn(120);     kame2.turn(120);
kame.goahead(100);  kame2.goahead(100);
```

つまり，左右の列にそれぞれの正三角形を描く命令を並べて書くのである．このように書くと，このプログラムの構造がプログラム２－３－２と同じであることがわかる．このことは，クラスから生成されるオブジェクトの仕組みを考えれば当然のことである．１つ１つのオブジェクトに対する共有メモリのイメージが図２－２のようになり，基本データに対応する局所変数を必要なだけ定義することと，基本データのコピーがオブジェクトごとに確保されることとが同じことであると考えられるからである．しかし，プログラムの表現性の点からみると，基本データが引数でなく，その個数が少ないことから，このプログラムの方がわかりやすいのは明らかである．また，どちらの三角形の処理かを判断するのに，プログラム２－３－２は引数の変数名からであり，このプログラムではオブジェクト名からであることによる違いがある．オブジェクト名までを関数名と考えれば，このプログラムの方がわかりやすいプログラムといえる．（例えば，２つのオブジェクトに対する変数名を kame_left と kame_right とすれば，よりわかりやすくできる．）また，形式的に同じプログラムとなる両者の拡張性の点からの比較では，基本データが隠されて表面にでてこないオブジェクト指向プログラミングの方が，優れていることは前述したとおりである．

ここで，オブジェクト指向プログラミングどうしのプログラム２－２とプログラム２－４を

比較してみる．表現性の点においては上で見たようにほとんど同じである．基本データの変更などによる拡張性は，クラス定義が同じであるから両者は変わらない．2個の正三角形が3個，4個に増えるというような処理の変更も，一つ一つに対応するオブジェクトを生成することで容易に対応できる．同時に，このことから，オブジェクト指向プログラミングが並行処理への応用に適していることも窺える．これは，大いに注目すべき点である．「2つの図形を描くタートル（ロボット）を用意し，それらに個別に命令を与えることによって，それぞれの正三角形を描く」という，この処理方法が自然でいかに現実を写像したものであるかは，1つ正三角形を描くときよりもはっきりとわかる．

次に，2つの正三角形を図2－3の番号順ではなく，左のものを描いた後，右のものを描くという手順で，それらを一つのオブジェクトで描くこと考えてみる．基本関数の中に初期設定以外のタートルを位置づける関数が必要となる．このために，タートルの位置と向きを同時に設定する関数 set_xya，タートルの位置だけを設定する関数 set_xy，タートルの向きだけを設定する関数 set_angle を，メンバ関数として次のように追加する．（実際には，関数 set_xya のみ必要であるが，拡張性を考えて他の関数も追加してある．）

```
class Turt {
    ........
    void turn(float);
    void set_xya(float, float, float);
    void set_xy(float, float);
    void set_angle(float);
}
    ........
void Turt :: set_xya(float x, float y, float a)  {X=x;  Y=y; Angle=a; }
void Turt :: set_xy(float x, float y)  {X=x;  Y=y;}
void Turt :: set_angle(float a)  {Angle=a; }
```

この処理を行うプログラムをプログラム2－4－2に示す．クラスのメンバ関数が増えただけで，基本的にはプログラム2－2と同じである．オブジェクトを1つにするか2個以上にするかは，現実の世界においてわれわれがどのような方法で正三角形を描くかを考えて決めればよい．言い方を変えれば，このクラス Turt は，現実の世界におけるタートル(軌跡によって図形を描く亀）の概念をうまくモデル化していることになる．

以上みてきたように，クラス Turt を使ったオブジェクト指向プログラミングの方が，表現性と拡張性の点において優位である．これは，クラスのカプセル化や情報隠蔽などの基本機能による点が大きいが，クラス Turt がタートル・グラフィックスというプログラムの概念をうまくモデル化していることにもよる．オブジェクト指向プログラミングでは，タートルの動きで図形を描くために必要な道具をクラスで用意し，利用者はその道具を使い現実の世界で描く時のイメージでプログラミングすれば，目的の図形が描けることになる．

```
// 2つの正三角形 (turt23.c)
#include <stdio.h>
#include <graphics.h>
#include <math.h>
double PI=3.14159265;
struct Tdata {
        float X,Y,Angle;
} turt;
void initurt(float x, float y, float a)
{
        turt.X=x; turt.Y=y; turt.Angle=a;
}
void goahead(float d)
{
        moveto(turt.X,turt.Y);
        turt.X+=d*cos(PI*turt.Angle/180);
        turt.Y-=d*sin(PI*turt.Angle/180);
        lineto(turt.X,turt.Y);
}
void turn(float a)
{
        turt.Angle+=a;
}
void main(void)
{
        int gdrive=DETECT,gmode;
        struct Tdata T1,T2;
        initgraph(&gdrive,&gmode,"a:¥¥TC¥¥BGI");
        cleardevice();
        setcolor(BLUE);
        initurt(100,200,0);
        goahead(100);
        turn(120);
        T1=turt;
        initurt(320,200,0);
        goahead(100);
        turn(120);
        T2=turt;
        initurt(T1.X,T1.Y,T1.Angle);
        goahead(100);
        turn(120);
        T1=turt;
        initurt(T2.X,T2.Y,T2.Angle);
        goahead(100);
        turn(120);
        T2=turt;
        initurt(T1.X,T1.Y,T1.Angle);
        goahead(100);
        initurt(T2.X,T2.Y,T2.Angle);
        goahead(100);
        getch();
        closegraph();
}
```

プログラム２－３　２つの正三角形（手続き型１）

```
// 2つの正三角形 (turt232.c)
#include <stdio.h>
#include <graphics.h>
#include <math.h>
double PI=3.14159265;
void goahead(float* X, float* Y, float Angle, float d)
{
        moveto(*X,*Y);
        *X+=d*cos(PI*Angle/180);
        *Y-=d*sin(PI*Angle/180);
        lineto(*X,*Y);
}
void turn(float* Angle, float a)
{
        *Angle+=a;
}
void main(void)
{
        int gdrive=DETECT,gmode;
        struct Tdata {
                float X,Y,Angle;
        } T1,T2;
        initgraph(&gdrive,&gmode,"a:¥¥TC¥¥BGI");
        cleardevice();
        setcolor(BLUE);
        T1.X=100; T1.Y=200; T1.Angle=0;
        goahead(&T1.X,&T1.Y,T1.Angle,100);
        turn(&T1.Angle,120);
        T2.X=320; T2.Y=200; T2.Angle=0;
        goahead(&T2.X,&T2.Y,T2.Angle,100);
        turn(&T2.Angle,120);
        goahead(&T1.X,&T1.Y,T1.Angle,100);
        turn(&T1.Angle,120);
        goahead(&T2.X,&T2.Y,T2.Angle,100);
        turn(&T2.Angle,120);
        goahead(&T1.X,&T1.Y,T1.Angle,100);
        goahead(&T2.X,&T2.Y,T2.Angle,100);
        getch();
        closegraph();
}
```

プログラム２－３－２　２つの正三角形（手続き型２）

```
// 2つの正三角形 (turt24.cpp)
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>
const double PI=3.14159265;
class Turt {
  private:
        float X,Y,Angle;
  public:
        Turt();
        Turt(float,float,float);
        void goahead(float);
        void turn(float);
```

```
};
Turt::Turt()
{
        X=320; Y=200; Angle=0;
}
Turt::Turt(float x, float y, float a)
{
        X=x; Y=y; Angle=a;
}
void Turt::goahead(float d)
{
        moveto(X,Y);
        X+=d*cos(PI*Angle/180);
        Y-=d*sin(PI*Angle/180);
        lineto(X,Y);
}
void Turt::turn(float a)
{
        Angle+=a;
}
void main(void)
{
        int gdrive=DETECT,gmode;
        initgraph(&gdrive,&gmode,"a:¥¥TC¥¥BGI");
        Turt kame(100,200,0),kame2(320,200,0);
        cleardevice();
        setcolor(BLUE);
        kame.goahead(100);
        kame.turn(120);
        kame2.goahead(100);
        kame2.turn(120);
        kame.goahead(100);
        kame.turn(120);
        kame2.goahead(100);
        kame2.turn(120);
        kame.goahead(100);
        kame2.goahead(100);
        getch();
        closegraph();
}
```

プログラム２－４　２つの正三角形（オブジェクト指向１）

```
// ２つの正三角形 (turt242.cpp)
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>
const double PI=3.14159265;
class Turt {
  private:
        float X,Y,Angle;
  public:
        Turt();
        Turt(float,float,float);
        void goahead(float);
        void turn(float);
```

```
        void set_xya(float,float,float);
        void set_xy(float,float);
        void set_angle(float);
};
Turt::Turt()
{
        X=320; Y=200; Angle=0;
}
Turt::Turt(float x, float y, float a)
{
        X=x; Y=y; Angle=a;
}
void Turt::goahead(float d)
{
        moveto(X,Y);
        X+=d*cos(PI*Angle/180);
        Y-=d*sin(PI*Angle/180);
        lineto(X,Y);
}
void Turt::turn(float a)
{
        Angle+=a;
}
void Turt::set_xya(float x, float y, float a)
{
        X=x; Y=y; Angle=a;
}
void Turt::set_xy(float x, float y)
{
        X=x; Y=y;
}
void Turt::set_angle(float a)
{
        Angle=a;
}
void main(void)
{
        int gdrive=DETECT,gmode;
        initgraph(&gdrive,&gmode,"a:¥¥TC¥¥BGI");
        Turt kame(100,200,0);
        cleardevice();
        setcolor(BLUE);
        kame.goahead(100);
        kame.turn(120);
        kame.goahead(100);
        kame.turn(120);
        kame.goahead(100);
        kame.set_xya(320,200,0);
        kame.goahead(100);
        kame.turn(120);
        kame.goahead(100);
        kame.turn(120);
        kame.goahead(100);
        getch();
        closegraph();
}
```

プログラム２―４―２　２つの正三角形（オブジェクト指向２）

3．クラスの継承と派生クラス

クラスの継承（inheritance）は，前章で述べたプログラミングパラダイムにもあるように，オブジェクト指向プログラミングを支える概念の一つである．一つの完成されたクラスを変更（メンバの追加，修正など）して目的に合うクラスを作り上げることができれば，データのカプセル化の有用性が高まるだけではなく拡張性が高まる．C＋＋では，既存のクラスから目的に合うクラスを作成するために継承を使う．継承の基になるクラスを基底クラス（上位クラスとか親クラスとも呼ぶ），それから導出されるクラスを派生クラス（下位クラスとか導出クラスとも呼ぶ）と呼ぶ．ここでは，前章に引き続きタートル・グラフィックスを例にとり，手続き型プログラミングとオブジェクト指向プログラミングを比較し，両者のプログラムの表現性と拡張性についての検証を行うが，特にクラスの継承とそれらとの関係について考察する．

図３－１のように左の正三角形を青色で，右の正三角形を黄色で描くプログラムについて考える．ただし，書く順序は図２－３と同じで，色指定をする変数を基本データとし，この基本データに値を設定する基本関数を追加するものとする．手続き型プログラミングで表現したものがプログラム３－１，オブジェクト指向プログラミングで表現したものがプログラム３－２である．

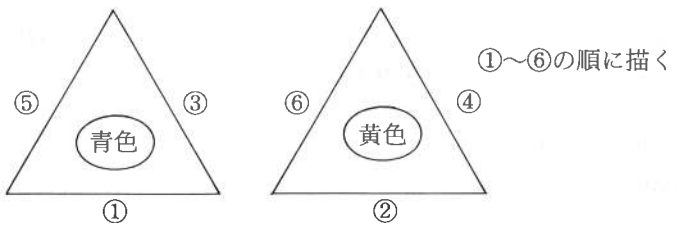

図３－１　色指定の２つの正三角形

手続き型プログラミング（プログラム３－１）では，色指定の基本データの変数 Color を構造体 Tdata に追加し，その変数に値を設定する基本関数を関数 set_color で定義している．さらに，色指定してタートルの軌跡を描く関数 goahead_c を，次のような基本関数として追加している．

```
void goahead_c(float d)
{
    setcolor(Color);
    goahead(d);
}
```

追加したこの関数を使い，プログラム２－３を修正して作成している．主な変更点は，関数 goahead の代わりに関数 goahead_c を呼び出すことと，この関数を呼びだす直前に関数 set_color を呼びだして，変数 Color に指定の色を設定していることである．プログラム２－３と比較して，色指定の処理が増えた分だけプログラムがわかりにくくなっている．このように，

プログラムの機能の追加に伴いプログラムが複雑になることは，前述したように手続き型プログラミングでは一般的である．既存のプログラムを修正して作成するより，最初から全体を作成した方がよい場合も多い．したがって，このプログラムは，表現性の点からも拡張性の点からも良いとはいえない．なお，関数 goahead_c の代わりに，次のように関数 goahead を変更して使う方法も考えられる．

```
void goahead(float d)
{
    setcolor(Color);
     (以下，プログラム２－３と同じ)
}
```

しかし，このような変更は，関数 goahead の機能の変更になり，既存のプログラム（例えばプログラム２－１やプログラム２－３）との関係を考えれば良くないことは明らかである．タートル・グラフィックスの基本関数で，機能の異なる同じ関数名のものが２つ存在することになり，これは利用者に混乱を招いてしまうし，それ以上にこれらの管理が難しいからである．

オブジェクト指向プログラミング（プログラム３－２）では，クラスの継承を利用し，クラス Turt から次のようなクラス Turt_c を導出している．

```
class Turt_c : public Turt {
    private :
        int Color ;
    public :
        Turt_c();
        Turt_c(float, float, float, int);
        void set_color(int);
        void goahead_c(float);
};
```

クラス Turt が基底クラス，クラス Turt_c が派生クラスである．この派生クラス Turt_c から次のようにして２つのオブジェクトを生成している．

```
Turt_c kame(100, 200, 0, BLUE), kame 2(320, 200, 0, YELLOW);
```

図３－１の左右の正三角形を描くオブジェクトが，それぞれ kame と kame 2 である．この定義によって，クラス Turt の基本データ（X，Y，Angle）とクラス Turt_c の基本データ（Color）のコピーがそれぞれのオブジェクトに確保され，それぞれの引数の値によって初期化される．また，クラス Turt の public 部の基本関数とクラス Turt_c の基本関数が，これらのオブジェクトに対するメッセージとして有効となる．例えば，オブジェクト kame に対する次のような

メッセージである．

```
kame.goahead_c(100);
kame.turn(120);
```

基本関数 goahead_c はクラス Turt_c で，基本関数 turn はクラス Turt で定義されたものである．また，基本関数 goahead_c は，基底クラス Turt の基本関数 goahead を呼びだして次のように定義されている．（これはプログラム３－１と同じである．）

```
void Turt_c::goahead_c(float d)
{
    setcolor(Color);
    goahead(d);
}
```

基底クラスで定義された public 部の基本関数は，派生クラスの基本関数の中からでも呼びだして使うことができる．C＋＋の継承では，派生クラスに基底クラスのすべてのデータメンバのコピーが確保され，派生クラスから基底クラスの public 部と protected 部のすべてのメンバを参照することができる．特に，protected 部は，派生クラスからの参照を前提に考えたメンバを定義するところで，継承のためのクラスの基本機能の一つである．

プログラム２－４と比較して，main 関数の部分はほとんど変わっていない．変わっている点は，オブジェクトの型が Turt_c で定義されていることと，オブジェクトに対するメッセージが goahead ではなく goahead_c になっていることである．また，プログラム２－４の所でも述べたように，main 関数の各オブジェクトへのメッセージの部分を次のように書けば，どちらの正三角形を描くための処理なのかがよりわかりやすくなる．

```
kame.goahead_c(100); kame2.goahead_c(100);
kame.turn(120);      kame2.turn(120);
kame.goahead_c(100); kame2.goahead_c(100);
kame.turn(120);      kame2.turn(120);
kame.goahead_c(100); kame2.goahead_c(100);
```

プログラムの機能の追加により，手続き型プログラミングではプログラムがわかりにくくなってしまうが，オブジェクト指向プログラミングではプログラムのわかりやすさが変わらない．これは，クラスの継承をうまく利用できたからであり，オブジェクト指向プログラミングでのクラスの継承がいかに有効であるかを示している．クラスの継承によりクラスの階層構造ができるが，これにはクラスの特殊化，汎用化，抽象化などの概念がある[10]．このプログラムのクラス Turt_c は，クラス Turt を特殊化したものといえる．つまり，クラス Turt のタートルが単に正三角形を描くタートルであるのに対し，クラス Turt_c のそれは色までもつけてくれる特殊なタートルであるということである．このように考えると，プログラム３－２では，普

通のタートルではなく色つきの特殊なタートルを用意し，それに対する命令であるから色つきの正三角形が描かれるということになる．もし，色指定しない正三角形と色指定する正三角形を同時に描きたいのであれば，それぞれのタートルをクラス Turt とクラス Turt_c のオブジェクトとして用意し，それらに命令を与えるだけでよい．プログラムの表現もプログラム３－２と同じである．これらの変更が容易であることから考えて，プログラム３－２はかなり拡張性の高いプログラムといえる．このような継承を利用したクラスの階層構造によって，プログラムの概念を的確に表現できることがオブジェクト指向プログラミングのプログラミング方法であり，そのようにすることによってわかりやすく拡張性のあるプログラムが作成できるといえる．

また，クラスの継承では，基底クラスのデータメンバは変更できないが，メンバ関数は再定義できるようになっている．したがって，クラス Turt_c のメンバ関数 goahead_c の関数名を基底クラスのメンバ関数 goahead と同じものとして，次のように定義することもできる．

```
void Turt_c::goahead(float d)
{
    setcolor(Color);
    以下，クラス Turt のメンバ関数 goahead と同じ
}
```

このように定義しても，基底クラスの利用者には全く影響しない．なぜならば，この変更はクラスの継承によるものであり，基底クラスの変更ではないからである．この点が，プログラム３－１の場合と大きく違うところであり，クラスの継承の有効性を示している．

以上見てきたように，プログラムの機能の追加により main 関数においては，手続き型プログラミングではプログラムがわかりにくくなってしまうが，オブジェクト指向プログラミングではわかりやすさの点では追加前のプログラムと変わらない．これは，実質的なプログラムの変更がクラスの変更だけですむからである．その変更もクラスの継承により簡単にできた．このことは，オブジェクト指向プログラミングの拡張性の高さも示している．一方，手続き型プログラミングでは，プログラムの変更が基本関数と main 関数に及ぶことから拡張性があるとはいえない．オブジェクト指向プログラミングにおいては，基底クラスにはないデータメンバやメンバ関数を追加したり，基底クラスにあるメンバ関数を修正したりして，用途にあった派生クラスを作成することで，プログラムの拡張性（再利用性）が高まるものと考えられる．

```
// 色指定の２つの正三角形 (turt31.c)
#include <stdio.h>
#include <graphics.h>
#include <math.h>
double PI=3.14159265;
struct Tdata {
        float X,Y,Angle,Color;
} turt;
void initurt(float x, float y, float a)
{
        turt.X=x; turt.Y=y; turt.Angle=a;
}
void goahead(float d)
```

```
{
        setcolor(turt.Color);
        moveto(turt.X,turt.Y);
        turt.X+=d*cos(PI*turt.Angle/180);
        turt.Y-=d*sin(PI*turt.Angle/180);
        lineto(turt.X,turt.Y);
}
void turn(float a)
{
        turt.Angle+=a;
}
void set_color(int c)
{
        turt.Color=c;
}
void main(void)
{
        int gdrive=DETECT,gmode;
        struct Tdata T1,T2;
        initgraph(&gdrive,&gmode,"a:¥¥TC¥¥BGI");
        cleardevice();
        initurt(100,200,0);
        set_color(BLUE);
        goahead(100);
        turn(120);
        T1=turt;
        initurt(320,200,0);
        set_color(YELLOW);
        goahead(100);
        turn(120);
        T2=turt;
        initurt(T1.X,T1.Y,T1.Angle);
        set_color(T1.Color);
        goahead(100);
        turn(120);
        T1=turt;
        initurt(T2.X,T2.Y,T2.Angle);
        set_color(T2.Color);
        goahead(100);
        turn(120);
        T2=turt;
        initurt(T1.X,T1.Y,T1.Angle);
        set_color(T1.Color);
        goahead(100);
        initurt(T2.X,T2.Y,T2.Angle);
        set_color(T2.Color);
        goahead(100);
        getch();
        closegraph();
}
```

プログラム３－１　色指定の２つの正三角形（手続き型）

```
// 色指定の２つの正三角形（turt32.cpp）
#include <iostream.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>
```

```
const double PI=3.14159265;
class Turt {
  private:
        float X,Y,Angle;
  public:
        Turt();
        Turt(float,float,float);
        void goahead(float);
        void turn(float);
};
Turt::Turt()
{
        X=320; Y=200; Angle=0;
}
Turt::Turt(float x, float y, float a)
{
        X=x; Y=y; Angle=a;
}
void Turt::goahead(float d)
{
        moveto(X,Y);
        X+=d*cos(PI*Angle/180);
        Y-=d*sin(PI*Angle/180);
        lineto(X,Y);
}
void Turt::turn(float a)
{
        Angle+=a;
}
class Turt_c : public Turt {
  private:
        int Color;
  public:
        Turt_c();
        Turt_c(float,float,float,int);
        void set_color(int);
        void goahead_c(float);
};
Turt_c::Turt_c() : Turt()
{
        Color=WHITE;
}
Turt_c::Turt_c(float x=320, float y=200, float a=200, int c=WHITE)
        : Turt(x,y,a)
{
        Color=c;
}
void Turt_c::set_color(int c)
{
        Color=c;
}
void Turt_c::goahead_c(float d)
{
        setcolor(Color);
        goahead(d);
}
void main(void)
{
```

```
        int gdrive=DETECT,gmode;
        initgraph(&gdrive,&gmode,"a:¥¥TC¥¥BGI");
        Turt_c kame(100,200,0,BLUE),kame2(320,200,0,YELLOW);
        cleardevice();
        kame.goahead_c(100);
        kame.turn(120);
        kame2.goahead_c(100);
        kame2.turn(120);
        kame.goahead_c(100);
        kame.turn(120);
        kame2.goahead_c(100);
        kame2.turn(120);
        kame.goahead_c(100);
        kame2.goahead_c(100);
        getch();
        closegraph();
}
```

プログラム３－２　色指定の２つの正三角形（オブジェクト指向）

４．演算子の多重定義

オブジェクト指向プログラミングの大きな目標は，問題を自然に近い形で表現するコードを作ることである[11]．C＋＋の関数や演算子の多重定義（overloading）は，その目標を具体的に実現するための機能の一つである．手続き型プログラミングでは同じ関数名の定義はできないが，オブジェクト指向プログラミングではシグネチュア（signature）[12]が異なれば同じ関数名や演算子記号を多重定義できる．例えば，次のような関数はすべて違う関数として多重定義できる．

```
void func1(int x);
void func1(float x);
int func2(int x);
int func2(int x, int y);
```

この多重定義を有効に利用することによって，わかりやすく拡張性のあるプログラムを作成することができる．例えば，有理数や複素数などをプログラムで扱う場合，それらのクラスで演算子記号（＋，－など）を多重定義することによって，既存の型（整数型や実数型など）と同じような演算式が書けるようになる．これもオブジェクト指向プログラミングの大きな特徴の一つといえる．ここでは，有理数クラスを例にとり，関数呼び出しの表現となる手続き型プログラミングとの比較により，演算子の多重定義によるプログラムの表現性と拡張性について検証し，その有効性について考察する．

有理数のプログラムでは，有理数を分子と分母の２つの整数（倍精度整数型）の組で表現し，それに対する次のような演算や処理を行う関数を用意するものとする．（以下，これらの関数を有理数基本関数と呼ぶ．）

・加算　（＋）
・減算　（－）

・乗算　（＊）
・除算　（／）
・比較　（＝＝）
・有理数の約分
・実数値（有理数を実数値に変換）
・有理数の表示

手続き型プログラミングで表現したものがプログラム４－１である．まず，有理数の分子と分母の組を構造体 Rdata（分子 Nu，分母 De）で表現し，有理数基本関数を次のような関数で定義している．

・加算	R_add(Rdata r 1, Rdata r 2)
・減算	R_sub(Rdata r 1, Rdata r 2)
・乗算	R_mul(Rdata r 1, Rdata r 2)
・除算	R_div(Rdata r 1, Rdata r 2)
・比較	R_equal(Rdata r 1, Rdata r 2)
・有理数の約分	R_simplify(Rdata r)
・実数値	R_real(Rdata r)
・有理数の表示	R_disp(Rdata r)

関数名の頭に R_をつけてあるのは，他の関数名と重複しないようにするためである．これらの関数を使った main 関数の内容について見てみる．まず，Rdata 型の初期値を持った３つの有理数 a(＝2/3)，b(＝2/-4)，c(＝4/6) と演算結果を代入する変数 ans を定義している．これらの変数を使った演算や処理は，すべて次のような関数呼び出しの形式で処理している．

関数名（引数１，引数２，・・・・，引数ｎ）；

例えば，a＋b，a＝＝b　の処理は次のようになる．

```
R_add(a, b);        (a+b)
R_equal(a, b)       (a==b)
```

これらは手続き型プログラミングの一般的な表現である．したがって，どのような処理をしているかは，関数名と引数の組み合わせから判断しなければならない．例のような簡単な計算式の場合はそれほどわかりにくいとはいえないが，「a ＊ b＋c」，「a－b ＊ (c＋a)」などの表現は，次のように複雑になってしまう．

```
R_add(R_mul(a, b), c);          (a * b+c)
R_sub(a, R_mul(b, R_add(c, a)));  (a-b * (c+a))
```

これ以上の複雑な計算式は，分割して表現した方がわかりやすくなる．しかし，分割して表現したものを読み直すのは，逆にわかりにくくなる．関数呼び出しの表現がいかに不自然かは明らかである．

また，「a＝2/3」と表示するには次のような表現となる．

```
cout ＜＜ "a＝";  R_disp(a);
```

関数 R_disp の処理内容があらかじめわかっていないと，この表現の意味はわからない．これが次のような表現であれば，どのような意味かがはっきりわかる．

```
cout ＜＜ "a =" ＜＜ a;
```

このように手続き型プログラミングの表現は，関数呼び出しの形が中心となり，数学的な表現に比べれば不自然でわかりにくい．これらを自然な表現にできるのが，オブジェクト指向プログラミングの演算子記号の多重定義である．これらについて，プログラム４－１とプログラム４－２の比較の中で見ていく．なお，プログラム４－１の出力結果は次のようになる．

```
a ＝2/3 ，  b ＝2/－4 ，  c ＝4/6；
約分後の各分数の値
a ＝2/3 ，  b ＝－1/2 ，  c ＝2/3；
a ＋ b ＝1/3
a － b ＝7/6
a ＊ b ＝－1/3
a / b ＝－4/3
a ＊ c ＋ b ＝－1/18
a ＋ c ＊ b ＝1/3
a ！＝b
a ＝＝c
a ＝0.66667
```

オブジェクト指向プログラミングで表現したものがプログラム４－２である．まず，有理数クラス Ratio を定義し，private 部で分子 Nu と分母 De を定義している．有理数基本関数は，有理数の表示関数以外は次のようにクラスのメンバ関数で定義している．

- 加算　　operator ＋(Ratio r)
- 減算　　operator －(Ratio r)
- 乗算　　operator ＊(Ratio r)
- 除算　　operator / (Ratio r)
- 比較　　operator＝＝(Ratio r)
- 有理数の約分　　simplify(void)

・実数値　　operator()(void)
・有理数の表示　　(friend) operator<<(ostream & s, Ratio& r)

有理数の約分の関数は公開する必要がないのでprivate部の関数として，他の関数はすべてpublic部で演算子記号として定義している．有理数の表示関数は有理数クラスのフレンド関数でありメンバ関数ではない．(フレンド関数については後述する．) 演算子記号Xの定義は，関数名が頭にoperatorという文字をつけたoperatorXとなる以外は，普通の関数の定義と同じである．プログラム4－1の対応する関数と比較して，有理数の表示関数以外は引数の個数が一つだけ少なくなっている．これは，メッセージを受け取ったオブジェクトがもう一つの暗黙の引数となるからである．これら以外に次のような3つのコンストラクタが多重定義されている．

```
Ratio()
Ratio(long num)
Ratio(long num, long den)
```

これらのコンストラクタも，演算式を簡潔に書けるための重要な役割を果たしている．これについては後述する．

次に，main関数の内容について見ていく．このmain関数の処理は，プログラム4－1の約分後の有理数の表示の処理を除いたものと全く同じ内容のものである．したがって，このプログラムの出力結果は，プログラム4－1の最初の2行を除いたものと同じである．このmain関数の内容を見ると，プログラム4－2の表現が，いかに簡潔でわかりやすいものとなっているかは明らかである．言い換えれば，数学的に自然な表現となっている．これだけ見ても，オブジェクト指向プログラミングの方が，手続き型プログラミングよりも表現性の点において優位であることがわかる．以下，このような表現性の優位について詳しく考察する．

まず，クラスRatioのオブジェクトとして，3つの有理数a(=2/3)，b(=2/−4)，c(=4/6)を定義していることについてである．この定義では，分子と分母の値を引数とした3番目のコンストラクタによってオブジェクトが初期化される．したがって，いずれも既約分数の値となる．プログラム4－1では，定義した有理数を約分する必要があった．もし，既約分数でないものがあるとすれば，データに一貫性がなくなり，例えば比較関数(R_equal)は正しく動作しなくなってしまう．これを未然に防ぐためには，既約分数でないデータがあることを前提とした処理をしなければならない．もちろん，そのようなデータが処理結果に影響する関数についてである．しかし，データに一貫性のないデータ処理が，それのあるデータ処理と比べて複雑になることは明らかである．したがって，クラスのコンストラクタは，データの一貫性を保つことはもちろん，プログラムの表現性の点からも重要な役目を果たしているといえる．

次に，演算子記号の多重定義について見てみる．多重定義できる演算子記号は，一部のものを除いた既存の演算子記号に限られている．また，演算子記号の優先順位は，既存のものと同じで変更することはできない．演算子記号は，その定義によって初めて意味を持ち，既存のものと同じものにする必要はない．例えば，＋を減算の意味に定義することができる．しかし，そのような定義は利用者を混乱させるだけでなく，プログラムのバグの原因となる．したがって，

演算子記号の意味はできるだけ既存のものと同じようにした方が良い．まず，加算の演算子記号＋を例にとり，その多重定義について見てみる．この多重定義は，有理数の加算を意味するようにしてある．したがって，この記号を用いて，２つの有理数ａとｂの加算を次のような中置記法で自然に表現できる．

```
a＋b
```

これは，operator＋を関数名とみた，次の表現と同じ意味である．

```
a.operator＋(b)
```

operator＋は普通のメンバ関数と同じようなオブジェクトへのメッセージであり，実際にこのような表現でも良い．しかし，中置記法に比べ不自然であるこのような表現を，敢えてする必要はない．また，この多重定義により次のような演算の表現もできる．

```
a＋1
```

これは「a.operator＋(1)」と同じことであるが，引数の型変換により定数１が自動的に有理数型に変換されるからである．この変換の時，２番目に定義されたコンストラクタが利用される．このように，コンストラクタは，オブジェクトの直接の初期化だけではなく間接的な部分での役割も持つ．プログラム４－１を修正して，これと同じような処理ができるようにするためには次のような関数の定義が必要となる．

```
R_add 2(Rdata r 1, long r 2);
```

このような関数の追加は，加算についてだけでなく他の演算についても同様である．また，関数名も考慮しなければならない．ここでは，単純にR_add 2としてあるが，このような関数名が利用者にとって不都合であるのは明らかである．関数R_addとの違いを考慮した関数名が必要である．このようなことを考えても，手続き型プログラミングが拡張性の点からもいかに問題があるかがわかる．

また，有理数の表示の演算子記号の多重定義も，プログラムを簡潔でわかりやすい表現にする点での役割は大きい．このことは，次のようにプログラム４－１のものと比較するだけではっきりする．

（プログラム４－２）

```
cout ＜＜ "a＋b＝" ＜＜ a＋b ＜＜ '¥n';
```

（プログラム４－１）

```
ans＝R_add(a,b); cout ＜＜ "a＋b＝"; R_disp(ans); cout ＜＜ '¥n';
```

または

```
cout ＜＜ "a＋b＝"; R_disp(R_add(a,b) ); cout ＜＜ '¥n';
```

このような簡潔な表現ができるのも，クラス ostream[13]とクラス Ratio における演算子記号 "<<" の多重定義による．

以上見てきたように，有理数の処理においては，クラスの演算子記号の多重定義やコンストラクタなどにより，オブジェクト指向プログラミングの方が，手続き型プログラミングよりも簡潔でわかりやすい表現ができることがわかった．また，拡張性の点からも一部ではあるが優位であることもわかった．ここで，拡張性の点からもう少しプログラム 4 — 2 を見直してみる．

プログラム 4 — 2 で定義されたクラス Ratio では，次のような演算の表現はできない．

```
1+a
```

これは「1.operator+(a)」と同じことになり，定数 1 はクラス Ratio のオブジェクトではなくエラーとなってしまうからである．これをできるようにしたのがプログラム 4 — 3 である．プログラム 4 — 3 とプログラム 4 — 2 との違いは，次のような関数をメンバ関数ではなく，有理数クラスのフレンド関数として定義していることである．

- 加算　(friend) operator +(Ratio r 1, Ratio r 2)
- 減算　(friend) operator −(Ratio r 1, Ratio r 2)
- 乗算　(friend) operator *(Ratio r 1, Ratio r 2)
- 除算　(friend) operator /(Ratio r 1, Ratio r 2)
- 比較　(friend) operator==(Ratio r 1, Ratio r 2)

ここでは，フレンド指定が重要な役目を果たすので，これについて簡単に説明しておく．フレンド関数は，クラスの public 部だけでなく private 部のメンバも参照できる関数であり，クラスの定義の中でそのプロトタイプだけ宣言すればよい．フレンドの指定は関数だけでなく，クラス単位での指定もできる．フレンド指定されたフレンドクラスのすべてのメンバ関数がフレンド関数となる．したがって，クラスの private 部のメンバを参照できるのは，クラスのメンバ関数とフレンド関数およびフレンドクラスのメンバ関数である．しかし，フレンドの指定は，情報隠蔽の機能と相反するものであり必要最小限にしなければならない．

プログラム 4 — 2 のメンバ関数とこれらのフレンド関数との違いは，引数の個数がそれぞれ一つだけ増えていることである．これは，メッセージを受け取ったオブジェクトの暗黙の引数がなくなったからである．機能的な違いはそれほどないが，前述したように次のような演算の表現ができることから，プログラム 4 — 3 のクラス定義の方が拡張性があるといえる．

```
1 +a
```

これは「operator+(1,a)」を意味するが，前述したように定数 1 が有理数型に自動的に変換されるからである．これと同じ機能を手続き型プログラミングのプログラム 4 — 1 に追加するためには，さらに次のような関数が必要となる．

```
R_add 3(long r 1, Rdata r 2);
```

このような関数の追加が拡張性の点から問題があることは，前述したとおりである．

このプログラムの main 関数の内容は，プログラム４－２のものと全く同じである．したがって，出力結果も全く同じである．

有理数クラスを既存のデータ型（整数型や実数型など）と同じような型に近づけるためには，上で定義した演算子記号の他に代入演算子（＋＝など）やインクリメント演算子（＋＋）などを追加する必要がある．また，出力だけでなく入力も分数の表現(例えば 1/3)でできるようにすれば，かなり便利である．プログラム４－３に，このような機能を追加したものがプログラム４－４である．追加した演算子は次のような関数である．

- 加算代入　　operator ＋＝(Ratio r)
- 減算代入　　operator －＝(Ratio r)
- 乗算代入　　operator ＊＝(Ratio r)
- 除算代入　　operator ／＝(Ratio r)
- インクリメント　operator＋＋(void)
- デクリメント　operator－－(void)
- 有理数の入力　(friend) operator>>(istream & s, Ratio r)

このような関数を有理数クラスに追加するだけで，目的の機能を追加できることから判断して，プログラム４－３は拡張性の高いものといえる．main 関数には，プログラム４－３の内容に，入力の処理と代入演算子などの処理を追加してある．入力および出力結果は次のようになる．

```
a(nu/de)入力⟶ 2/3
b(nu/de)入力⟶ 2/－4
c(nu/de)入力⟶ 4/6
a＝2/3 , b＝2/-4 , c＝4/6；
約分後の各分数の値
a＝2/3 , b＝－1/2 , c＝2/3；
a＋b＝1/3
a－b＝7/6
a ＊ b ＝－1/3
a ／ b ＝－4/3
a ＊ c ＋ b ＝－1/18
a ＋ c ＊ b ＝1/3
a ！＝b
a ＝＝c
a ＝0.66667
a ＋＝ b⟶＝1/6
a －＝ b⟶＝2/3
a ＊＝ b⟶＝－1/3
```

```
a / = b ⟶ =2/3
a ++ =5/3
a -- =2/3
c +2=8/3
2+ c =8/3
2 * a + b =5/6
(2 * a + b)/(c +2)=5/16
```

main 関数の内容とこれらの結果をみれば，プログラム４－４における有理数クラスの定義によって，有理数の演算が，既存の型と同じように自然な演算式で表現できることがわかる．なお，プログラム４－４で追加した代入演算子やインクリメントの処理を，プログラム４－１に追加することは意味がない．なぜなら，これらは演算式の表現だけの問題であるからである．

プログラム４－４のクラスの機能の内容からして，有理数クラスは既存の型とほぼ同じような演算式の表現ができる．また，新たな表現が必要となれば，有理数クラスにその処理を追加するだけでよい．この追加処理も，クラス単位の変更である限り，簡単に実現できると思われる．この有理数クラスは，一つの完成された利用者定義の新しい型である．このように一つの利用者定義の型として完成されたクラスは，既存の型として他のプログラムで有効に利用されるようになる．さらに，完成されたクラスでも，クラスの継承を使用することによって，他のプログラムに影響を及ぼさずに変更できる．これらのことは，手続き型プログラミングでは容易に実現できることではなく，表現性や拡張性の点において，それに対するオブジェクト指向プログラミングの優位性を示すものといえる．

```
// プログラム４－１(ratio41.cpp)
#include <iostream.h>
#include <math.h>
struct Rdata {
        long Nu;        // numeraor
        long De;        // denomirator
};
Rdata R_simplify(Rdata r)
{
        Rdata ans=r;
        if (r.Nu==0) { ans.De=1; return ans; }
        if (r.De<0)  { ans.Nu=-r.Nu; ans.De=-r.De; }
        long gcd=abs(ans.Nu), n=ans.De, rr;
        while (n!=0) {
                rr=gcd%n; gcd=n; n=rr;
        }
        ans.Nu=ans.Nu/gcd;
        ans.De=ans.De/gcd;
        return ans;
}
double real(Rdata r) {
        return double(r.Nu)/r.De;
}
Rdata R_add (Rdata r1,Rdata r2) {
```

```
        Rdata ans;
        ans.Nu=r1.Nu*r2.De + r1.De*r2.Nu;
        ans.De=r1.De*r2.De;
        return R_simplify(ans);
}
Rdata R_sub(Rdata r1,Rdata r2) {
        Rdata ans;
        ans.Nu=r1.Nu*r2.De - r1.De*r2.Nu;
        ans.De=r1.De*r2.De;
        return R_simplify(ans);
}
Rdata R_mul(Rdata r1,Rdata r2) {
        Rdata ans;
        ans.Nu=r1.Nu*r2.Nu;
        ans.De=r1.De*r2.De;
        return R_simplify(ans);
}
Rdata R_div(Rdata r1,Rdata r2) {
        Rdata ans;
        if (r2.Nu==0) {
                cerr << "分数の割り算エラー";
                ans.Nu=0;  ans.De=1;
                return ans;
        }
        ans.Nu=r1.Nu*r2.De;
        ans.De=r1.De*r2.Nu;
        return R_simplify(ans);
}
int R_equal(Rdata r1,Rdata r2) {
        Rdata sr1=R_simplify(r1);
        Rdata sr2=R_simplify(r2);
        if (sr1.Nu==sr2.Nu && sr1.De==sr2.De)
                return 1;
        else
                return 0;
}
void R_disp(Rdata r)
{
        if (r.De==1)
                cout << r.Nu;
        else
                cout << r.Nu << '/' << r.De;
}
void main(void)
{
        Rdata a={2,3},b={2,-4},c={4,6},ans;
        cout << "a=";    R_disp(a);
        cout << "  b=";  R_disp(b);
        cout << "  c=";  R_disp(c); cout << '¥n';
        cout << "約分後の各分数の値¥n";
        cout << "a=";    R_disp( R_simplify(a) );
        cout << "  b=";  R_disp( R_simplify(b) );
        cout << "  c=";  R_disp( R_simplify(c) ); cout << '¥n';
        ans=R_add(a,b);
        cout << "a+b=";  R_disp(ans); cout << '¥n';
        ans=R_sub(a,b);
        cout << "a-b=";  R_disp(ans); cout << '¥n';
        ans=R_mul(a,b);
```

```
        cout << "a*b=";  R_disp(ans); cout << '¥n';
        ans=R_div(a,b);
        cout << "a/b=";  R_disp(ans); cout << '¥n';
        ans=R_add( R_mul(a,c), b);
        cout << "a*c+b=";  R_disp(ans); cout << '¥n';
        ans=R_add( a, R_mul(c,b) );
        cout << "a+c*b=";  R_disp(ans); cout << '¥n';
        if ( R_equal(a,b) ) cout << "a==b¥n";  else cout << "a!=b¥n";
        if ( R_equal(a,c) ) cout << "a==c¥n";  else cout << "a!=c¥n";
        double rans=real(a);
        cout << "a=" << rans << "¥n";
}
```

プログラム４－１　有理数の処理（手続き型）

```
// プログラム４－２(ratio42.cpp)
#include <iostream.h>
#include <math.h>
class Ratio {
  private:
        long Nu;        // numeraor
        long De;        // denomirator
        void simplify(void);
  public:
        Ratio() { Nu=0; De=1; }
        Ratio(long num) { Nu=num; De=1; }
        Ratio(long num, long den) { Nu=num; De=den;  simplify(); }
        double operator()(void);
        Ratio operator+(Ratio);
        Ratio operator-(Ratio);
        Ratio operator*(Ratio);
        Ratio operator/(Ratio);
        int operator==(Ratio);
        friend ostream& operator<<(ostream&,Ratio&);
};
void Ratio::simplify(void)
{
        if (Nu==0) { De=1; return; }
        if (De<0)  { Nu=-Nu; De=-De; }
        long gcd=abs(Nu), n=De, r;
        while (n!=0) {
                r=gcd%n; gcd=n; n=r;
        }
        Nu=Nu/gcd;
        De=De/gcd;
}
double Ratio::operator()() {
        return double(Nu)/De;
}
Ratio Ratio::operator+(Ratio r) {
        long n_num=Nu*r.De + De*r.Nu;
        long n_den=De*r.De;
        return Ratio(n_num,n_den);
}
Ratio Ratio::operator-(Ratio r) {
        long n_num=Nu*r.De - De*r.Nu;
        long n_den=De*r.De;
```

```
        return Ratio(n_num,n_den);
}
Ratio Ratio::operator*(Ratio r) {
        long n_num=Nu*r.Nu;
        long n_den=De*r.De;
        return Ratio(n_num,n_den);
}
Ratio Ratio::operator/(Ratio r) {
        if ( r.Nu==0 ) { cerr << "分数の割り算エラー¥n"; return Ratio(0,1); }
        long n_num=Nu*r.De;
        long n_den=De*r.Nu;
        return Ratio(n_num,n_den);
}
int Ratio::operator==(Ratio r) {
        if (Nu==r.Nu && De==r.De)
                return 1;
        else
                return 0;
}
ostream& operator<<(ostream& s, Ratio& r)
{
        if (r.De==1)
                return( s << r.Nu );
        else
                return( s << r.Nu << '/' << r.De );
}
void main(void)
{
        Ratio a(2,3),b(2,-4),c(4,6);
        cout << "a=" << a << "  b=" << b << "  c=" << c << '¥n';
        cout << "a+b=" << a+b << '¥n';
        cout << "a-b=" << a-b << '¥n';
        cout << "a*b=" << a*b << '¥n';
        cout << "a/b=" << a/b << '¥n';
        cout << "a*c+b=" << a*c+b << '¥n';
        cout << "a+c*b=" << a+c*b << '¥n';
        if (a==b) cout << "a==b¥n";  else cout << "a!=b¥n";
        if (a==c) cout << "a==c¥n";  else cout << "a!=c¥n";
        cout << "a=" << a() << "¥n";
}
```

プログラム４－２　有理数の処理（オブジェクト指向１）

```
// プログラム４－３　(ratio43.cpp)
#include <iostream.h>
#include <math.h>
class Ratio {
  private:
        long Nu;        // numeraor
        long De;        // denomirator
        void simplify(void);
  public:
        Ratio() { Nu=0; De=1; }
        Ratio(long num) { Nu=num; De=1; }
        Ratio(long num, long den) { Nu=num; De=den;  simplify(); }
        double operator()(void);
        friend Ratio operator+(Ratio,Ratio);
```

```
        friend Ratio operator-(Ratio,Ratio);
        friend Ratio operator*(Ratio,Ratio);
        friend Ratio operator/(Ratio,Ratio);
        friend int operator==(Ratio,Ratio);
        friend ostream& operator<<(ostream&,Ratio&);
};
void Ratio::simplify(void)
{
        if (Nu==0) { De=1; return; }
        if (De<0)  { Nu=-Nu; De=-De; }
        long gcd=abs(Nu), n=De, r;
        while (n!=0) {
                r=gcd%n; gcd=n; n=r;
        }
        Nu=Nu/gcd;
        De=De/gcd;
}
double Ratio::operator()() {
        return double(Nu)/De;
}
Ratio operator+(Ratio r1,Ratio r2) {
        long n_num=r1.Nu*r2.De + r1.De*r2.Nu;
        long n_den=r1.De*r2.De;
        return Ratio(n_num,n_den);
}
Ratio operator-(Ratio r1,Ratio r2) {
        long n_num=r1.Nu*r2.De - r1.De*r2.Nu;
        long n_den=r1.De*r2.De;
        return Ratio(n_num,n_den);
}
Ratio operator*(Ratio r1,Ratio r2) {
        long n_num=r1.Nu*r2.Nu;
        long n_den=r1.De*r2.De;
        return Ratio(n_num,n_den);
}
Ratio operator/(Ratio r1,Ratio r2) {
        if ( r2.Nu==0 ) { cerr << "分数の割り算エラー¥n"; return Ratio(0,1);
}
        long n_num=r1.Nu*r2.De;
        long n_den=r1.De*r2.Nu;
        return Ratio(n_num,n_den);
}
int operator==(Ratio r1,Ratio r2) {
        if (r1.Nu==r2.Nu && r1.De==r2.De)
                return 1;
        else
                return 0;
}
ostream& operator<<(ostream& s, Ratio& r)
{
        if (r.De==1)
                return( s << r.Nu );
        else
                return( s << r.Nu << '/' << r.De );
}
void main(void)
{
```

```
        Ratio a(2,3),b(2,-4),c(4,6);
        cout << "a=" << a << "  b=" << b << "  c=" << c << '¥n';
        cout << "a+b=" << a+b << '¥n';
        cout << "a-b=" << a-b << '¥n';
        cout << "a*b=" << a*b << '¥n';
        cout << "a/b=" << a/b << '¥n';
        cout << "a*c+b=" << a*c+b << '¥n';
        cout << "a+c*b=" << a+c*b << '¥n';
        if (a==b) cout << "a==b¥n";  else cout << "a!=b¥n";
        if (a==c) cout << "a==c¥n";  else cout << "a!=c¥n";
        cout << "a=" << a() << "¥n";
}
```

プログラム４－３　有理数の処理（オブジェクト指向２）

```
// プログラム４－４(ratio44.cpp)
#include <iostream.h>
#include <math.h>
class Ratio {
  private:
        long Nu;        // numeraor
        long De;        // denomirator
        void simplify(void);
  public:
        Ratio() { Nu=0; De=1; }
        Ratio(long num) { Nu=num; De=1; }
        Ratio(long num, long den) { Nu=num; De=den;  simplify(); }
        Ratio& operator+=(Ratio);
        Ratio& operator-=(Ratio);
        Ratio& operator*=(Ratio);
        Ratio& operator/=(Ratio);
        Ratio& operator++();
        Ratio& operator--();
        double operator()(void);
        friend Ratio operator+(Ratio,Ratio);
        friend Ratio operator-(Ratio,Ratio);
        friend Ratio operator*(Ratio,Ratio);
        friend Ratio operator/(Ratio,Ratio);
        friend int operator==(Ratio,Ratio);
        friend ostream& operator<<(ostream&,Ratio&);
        friend istream& operator>>(istream&,Ratio&);
};
void Ratio::simplify(void)
{
        if (Nu==0) { De=1; return; }
        if (De<0)  { Nu=-Nu; De=-De; }
        long gcd=abs(Nu), n=De, r;
        while (n!=0) {
                r=gcd%n; gcd=n; n=r;
        }
        Nu=Nu/gcd;
        De=De/gcd;
}
Ratio& Ratio::operator+=(Ratio r) {
        Nu=Nu*r.De + De*r.Nu;
        De=De*r.De;
        simplify();
```

```
        return *this;
}
Ratio& Ratio::operator-=(Ratio r) {
        Nu=Nu*r.De - De*r.Nu;
        De=De*r.De;
        simplify();
        return *this;
}
Ratio& Ratio::operator*=(Ratio r) {
        Nu=Nu*r.Nu;
        De=De*r.De;
        simplify();
        return *this;
}
Ratio& Ratio::operator/=(Ratio r) {
        if ( r.De==0 ) {
                cerr << "分数の割り算エラー¥n";
                Nu=0; De=1;
                return *this;
        }
        Nu=Nu*r.De;
        De=De*r.Nu;
        simplify();
        return *this;
}
Ratio& Ratio::operator++() {
        Nu=Nu+De;
        return *this;
}
Ratio& Ratio::operator--() {
        Nu=Nu-De;
        return *this;
}
double Ratio::operator()(void){
        return double(Nu)/De;
}
Ratio operator+(Ratio r1,Ratio r2) {
        long n_num=r1.Nu*r2.De + r1.De*r2.Nu;
        long n_den=r1.De*r2.De;
        return Ratio(n_num,n_den);
}
Ratio operator-(Ratio r1,Ratio r2) {
        long n_num=r1.Nu*r2.De - r1.De*r2.Nu;
        long n_den=r1.De*r2.De;
        return Ratio(n_num,n_den);
}
Ratio operator*(Ratio r1,Ratio r2) {
        long n_num=r1.Nu*r2.Nu;
        long n_den=r1.De*r2.De;
        return Ratio(n_num,n_den);
}
Ratio operator/(Ratio r1,Ratio r2) {
        if ( r2.Nu==0 ) { cerr << "分数の割り算エラー¥n"; return Ratio(0,1);
}
        long n_num=r1.Nu*r2.De;
        long n_den=r1.De*r2.Nu;
        return Ratio(n_num,n_den);
}
```

```
int operator==(Ratio r1,Ratio r2) {
        if (r1.Nu==r2.Nu && r1.De==r2.De)
                return 1;
        else
                return 0;
}
ostream& operator<<(ostream& s, Ratio& r)
{
        if (r.De==1)
                return( s << r.Nu );
        else
                return( s << r.Nu << '/' << r.De );
}
istream& operator>>(istream& s, Ratio& r)
{
        long nu,de;
        char ch;
        s >> nu;
        s >> ch;
        if (ch!='/') { cerr << "入力エラー¥n"; return s; }
        s >> de;
        r=Ratio(nu,de);
        return s;
}
void main(void)
{
        Ratio a,b,c;
        cout << "a(nu/de)入力--> ";  cin >> a;    // a=2/3
        cout << "b(nu/de)入力--> ";  cin >> b;    // b=2/-4
        cout << "c(nu/de)入力--> ";  cin >> c;    // c=4/6
        cout << "a=" << a << "  b=" << b << "  c=" << c << '¥n';
        cout << "a+b=" << a+b << '¥n';
        cout << "a-b=" << a-b << '¥n';
        cout << "a*b=" << a*b << '¥n';
        cout << "a/b=" << a/b << '¥n';
        cout << "a*c+b=" << a*c+b << '¥n';
        cout << "a+c*b=" << a+c*b << '¥n';
        if (a==b) cout << "a==b¥n";  else cout << "a!=b¥n";
        if (a==c) cout << "a==c¥n";  else cout << "a!=c¥n";
        cout << "a=" << a() << "¥n";
        a+=b; cout << "a+=b-->=" << a << '¥n';
        a-=b; cout << "a-=b-->=" << a << '¥n';
        a*=b; cout << "a*=b-->=" << a << '¥n';
        a/=b; cout << "a/=b-->=" << a << '¥n';
        cout << "a++=" << a++  << '¥n';
        cout << "a--=" << a--  << '¥n';
        cout << "c+2=" << c+2  << '¥n';
        cout << "2+c=" << 2+c  << '¥n';
        cout << "2*a+b=" << 2*a+b << '¥n';
        cout << "(2*a+b)/(c+2)=" << (2*a+b)/(c+2) << '¥n';
}
```

プログラム４－４　有理数の処理（オブジェクト指向３）

5. おわりに

オブジェクト指向プログラミングと手続き型プログラミングで具体的なプログラムを作成し，両者をプログラムの表現性と拡張性の点で比較した結果，次のようなことがわかった．

(1) 1つの正三角形を描くプログラムでは同程度のわかりやすさであったが，指定された順で2つの正三角形を描くプログラムでは両者に大きな違いが見られた．手続き型プログラミングでは，単純な処理の追加に比べてかなりわかりにくいプログラムになってしまったが，オブジェクト指向プログラミングでは，1つの場合とそれほど変わらなかった．

(2) 2つの正三角形を描く場合に，指定した色で描くという機能を追加した処理においては，両者の間に変更のしかただけでなく変更後のプログラムに大きな違いが見られた．手続き型プログラミングでは，機能変更に伴うプログラムの変更がプログラム全体にわたり，そのことによってますますわかりにくいプログラムとなってしまった．オブジェクト指向プログラミングでは，機能変更に伴うプログラムの変更をクラスの変更に集約することができ，プログラム全体のわかりやすさは変わらなかった．しかも，クラスの変更も，継承の機能を有効に使うことによって簡単にできた．

(3) 有理数を処理するプログラムにおいては，関数呼び出し中心で表現する手続き型プログラミングに比べて，オブジェクト指向プログラミングでは演算子記号の多重定義により，既存の型と同じような演算式で有理数を扱えるようになり，数学的に自然でわかりやすいプログラムの表現ができた．また，演算機能の追加に対して，オブジェクト指向プログラミングはクラスの変更により柔軟に対応できたが，手続き型プログラミングではそれほど柔軟に対応できないことが確認できた．

以上の結果は，オブジェクト指向プログラミングの方が，手続き型プログラミングよりも表現性と拡張性の点で優位であることを示している．このような結果になったのは，オブジェクト指向プログラミングで確認できた次のような事項によるものと考えられる．

(1) クラスの持つデータのカプセル化と情報隠蔽の機能によりデータの一貫性が保てる．

(2) クラスの継承によりプログラムの機能変更に容易に対応できる．

(3) 手続き型プログラミングの関数呼び出しによる処理中心の命令に対し，オブジェクト指向プログラミングのオブジェクトに対する命令は，自然で現実の命令のイメージに近い．

(4) 演算子記号の多重定義により数学的に自然な演算式が表現できる．

本稿で確認できたオブジェクト指向プログラミングの優位性が，すべての領域に当てはまるとは言えない．例に挙げた領域が，たまたまオブジェクト指向プログラミングに適したものであったからかもしれないからである．さまざまな領域で両者のプログラムを比較し，どのような領域でオブジェクト指向プログラミングが有効であるかを調べる必要があると思われる．

注

(1) B. Stroustrup(1993) p.17
(2) 本稿では，Borland 社のC＋＋（AT&T のC＋＋バージョン 2.0 を実現している）を使用している．
(3) B. Stroustrup(1993) p.19
(4) P. Wegner(1990) p.1
(5) インライン関数は，呼びだされた位置にその関数本体が展開される関数である．
(6) 河西朝雄(1989) pp.252-259 を参考に作成
(7) 田中秀和(1992) pp.64-66 を参考に作成
(8) B. Stroustrup(1993) p.20
(9) B. Stroustrup(1993) p.29
(10) 深澤良彰,栗野俊一(1993) pp.69-92
(11) R. S. Wiener, L. J. Pinson(1988) p.89
(12) 関数名と引数（個数，型とその並び）から構成されるものを指す．
(13) B. Stroustrup(1993) p.431

参考文献

B. Stroustrup (1993)『The C++ Programming Language(Second Edition)』, Addison-Wesley
斎藤信男他訳 (1993)『プログラミングC＋＋(第2版)』, トッパン
R. S. Wiener, L. J. Pinson (1988)『An Introduction to Object-Oriented Programming and C++』, Addison-Wesley
前川守訳 (1989)『C＋＋：オブジェクト指向プログラミング』, トッパン
M. T. Skinner (1992)『The Advanced C++ Book』, Silicon Press
青木良且訳 (1994)『C＋＋実用講座』, インプレス
S. C. Dewhurst, K. T. Stark (1989)『Programming in C++』, Prentice-Hall
小山祐司監訳，ドキュメントシステム訳 (1990)『C＋＋言語入門』, アスキー
P. Wegner (1990)『Concepts and Paradigms of Object-Oriented Programmind』, ACM Press
尾内理紀夫訳 (1992)『はやわかりオブジェクト指向』, 共立出版
深澤良彰，栗野俊一 (1993)『オブジェクト指向プログラミング入門』, 共立出版
酒井博敬，堀内一 (1989)『オブジェクト指向入門』, オーム
田中秀和 (1992)『C＋＋によるプログラム設計法』, 総合電子出版
河西朝雄 (1989)『TURBO C 初級プログラミング上』, 技術評論社